# 动态规划要素

# 动态规划要素

- 什么样的优化问题适合使用**动态规划**?
  - ❖最优子结构　　分治，DP，Greedy
  - ❖重叠子问题　　DP is the best！

  利用重叠子问题特性可导出动态规划的变种方法：**memoization方法（记忆，备忘录）**

- **Wiki explanation:** In computing, memorization (a.k.a, memoisation) is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

# 最优子结构

- 若一个问题存在最优解，其内部包含的所有子问题解也必须最优，则该问题呈现了“最优子结构”，具有此结构特征的问题**可能**会使用动态规划。
  - 例如：矩阵$A_iA_{i+1}\dots A_j$的最优括号化问题蕴含着两个子问题$A_i\dots A_k$和$A_{k+1}\dots A_j$的解也必须是最优的。
  - 在动态规划中，可用子问题的最优解来构造原问题的最优解。

# 最优子结构

- 如何发现最优子结构呢(Principle)？
  - 说明问题的解必须进行某种选择，这种选择导致一个或多个待解的子问题。
  - 对一给定问题，假定导致最优解的选择已给定，即无须关心如何做出选择，只须假定它已给出。
  - 给定选择后，决定由此产生哪些子问题，如何最好地描述子问题结构(空间)
  - 证明用在问题最优解内的子问题的解也必须是最优的。方法是“*cut-and-paste*”技术和反证法。假定在最优解对应的子问题的解非最优，删去它换上最优解，得到原问题的解非最优，矛盾！

# 最优子结构

- 如何描述子问题空间（子问题结构，不同的子问题个数）

  尽可能使其简单，然后再考虑有没有必要扩展.

  例：$A_{1..n} \Rightarrow A_{1..k} \cdot A_{k+1..n} \Rightarrow (A_{1..k_1} \cdot A_{k_1+1..k})(A_{k+1..k_2} \cdot A_{k_2+1..n})$

  由此可见，最合适的子问题空间描述为：$A_i A_{i+1} \dots A_j$

- 最优子结构有关的两方面问题

  - 用在最优解中有多少个子问题（*i.e.*，每次做出选择会产生多少个子问题）
  - 用在最优解中的子问题有多少种选择

    例如，矩阵链乘 $A_i A_{i+1} \dots A_j$ ——两个子问题，$j - i$种选择

# 最优子结构

- 动态规划算法的运行时间
  - 子问题总数
  - 对每个子问题涉及多少种选择

  例如:

  矩阵链乘共要解$\Theta(n^2)$个子问题: $1 \le i \le j \le n$

  求解每个子问题至多有$n-1$种选择，最终的运行时间为$\Theta(n^3)$
- 动态规划求解方式
  - 自底向上

# 最优子结构

- **细节：不要随便假定最优子结构**

  例如有向无权图中，求最短/长路径的问题（指简单路径）

- **最短路径含有最优子结构**

  设从$u$到$v$的最短路径是$P$，并设中间点为$W$，则

  $$u \xrightarrow{P} v \quad \Rightarrow \quad u \xrightarrow{P_1} w \xrightarrow{P_2} v$$

  显然，$P_1$和$P_2$也必须是最优（短）的。

# 最优子结构

## 最长路径不具有最优子结构

设$P$是从$u$到$v$的最长路径，$W$是中间某点，则

$$u \xrightarrow{P} v \quad \Rightarrow \quad u \xrightarrow{P_1} w \xrightarrow{P_2} v$$

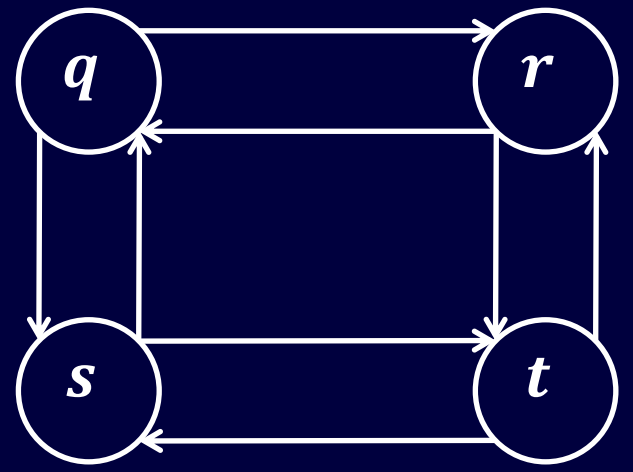
但$P_1$不一定是从$u$到$W$的最长路径，$P_2$也不一定是从$W$到$v$的最长路径。

例：

考虑一条最长路径：$q \to r \to t$

但$q$到$r$的最长路径是：$q \to s \to t \to r$

$r$到$t$的最长路径是：$r \to q \to s \to t$



# 最优子结构

- 为什么两问题有差别？
  - 最长路径的子问题不是独立的。所谓独立指一个子问题的解不能影响另一个子问题的解。但第一个子问题中使用了$s$和$t$，第二个子问题又使用了，使得产生的路径不再是简单路径。从另一个角度看：一个子问题求解时使用的资源(顶点)不能在另一个子问题中再使用。
  - 最短路径问题中，两子问题没有共享资源，可用反证法证明之。

    例：矩阵链乘，显然两子链不相交，无资源共享，是相互独立的两个子问题。

    $A_{i..j} \Rightarrow A_{i..k} \cdot A_{k+1..j}$

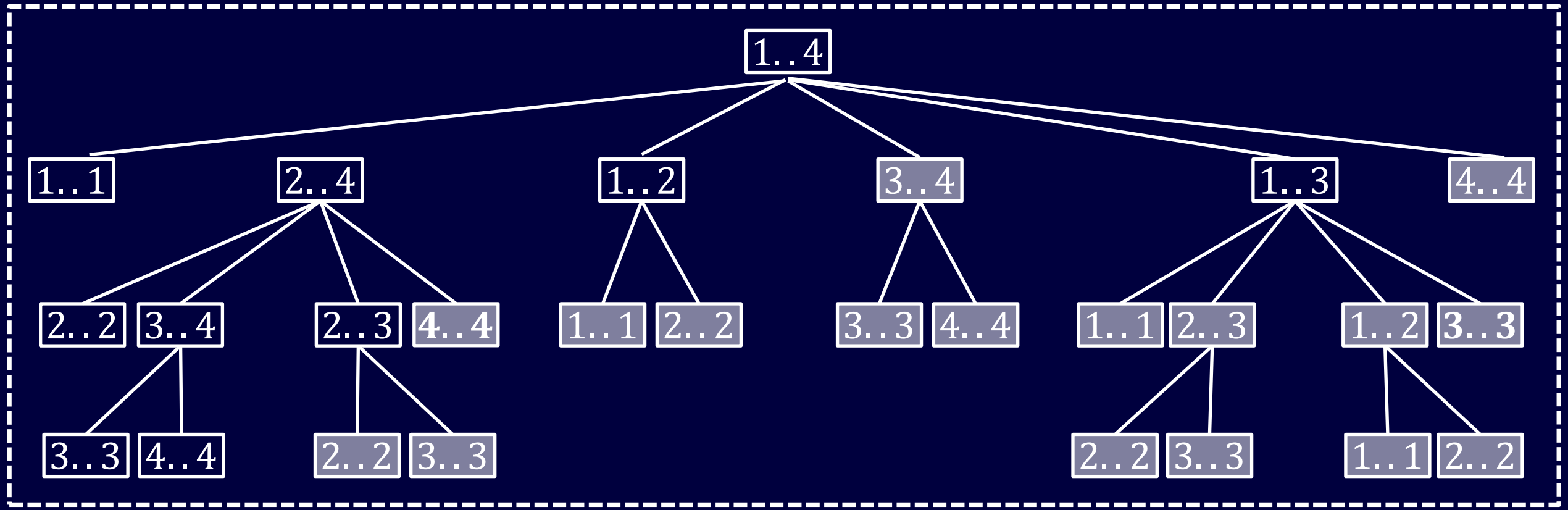
# 重叠子问题

- 当用递归算法解某问题时，重复访问(计算)同一子问题
- **分治法与动态规划的比较**
  - 当递归每一步产生一个**新**的子问题时，适合使用**分治法**。
  - **当递归中出现较多重叠子问题时，适合使用动态规划**，即对重叠子问题只求解一次，然后存储在表中，当需要使用时在常数时间内查表。**若子问题规模是多项式阶的，动态规划特别有效**。

# 重叠子问题

- 例：用自然递归算法求解

$$m[i,j] = \min_{1 \le k \le j-1} \{m[i,k] + m[k+1,j] + P_{i-1}P_kP_j\}$$



# 重叠子问题

- **阴影部分是重叠子问题**，递归算法须重复计算

  **递归式**时间：

$$\begin{cases} T(1) \geq 1 \\ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \end{cases} \quad \Rightarrow \quad T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

  用代入法可证为$\Omega(2^n)$

  **结论：**当自然递归算法是指数阶，但实际不同的子问题数目是多项式阶时，可用动态规划来获得高效算法。

# 重构最优解

- 用附加表保存中间选择结果能节省重构最优解时间。

  矩阵链乘法问题，如果只是在$m[i,j]$中记录子问题的最优代价，当我们确定了$A_iA_{i+1}\dots A_j$的最优括号化方案，重构这些最优子问题的时候，我们需检查所有$j-i$种可能，因此重构最优解时需要$\Theta(j-i)=\omega(1)$时间。而通过在表$S[i,j]$中保存$A_iA_{i+1}\dots A_j$的划分位置，我们重构每次选择只需要$O(1)$时间。

# 备忘录机制

- 动态规划：分析是自顶向下，实现是自底向上
- 备忘(记忆)型版本:它是动态规划的变种，效率和动态规划相似，但采用自顶向下实现，故是一个记忆型递归算法。和动态规划类似，将子问题的解记录在一个表中，但填表的控制结构更像递归算法，其特点是：
  - 每个子问题的解对应一表项。
  - 每个表目初值唯一，特殊值表示尚未填入。
  - 在递归算法执行过程中第一次遇某子问题时，计算其解并填入表中，以后再遇此子问题时，将表中值简单地返回(不重复计算)，截断递归。
- 该方法的前提
  - 原有可能的子问题参数集合是已知的
  - 可在表位置和子问题间建立某种关系

```
MEMOIZED_MATRIX_CHAIN(p)
    n = p.length − 1
    for i = 1 to n
        for j = i to n
            m[i,j] = ∞                    //表目初值，上三角
    return LOOKUP_CHAIN(m,p,1,n)

LOOKUP_CHAIN(m,p,i,j)
    if m[i,j] < ∞                         //已计算过
        return m[i,j]                     //截断递归
    if i == j
        m[i,j] = 0
    else
        for k = i to j − 1
            q = LOOKUP_CHAIN(m,p,i,k) + LOOKUP_CHAIN(m,p,k + 1,j) + p_{i−1} p_k p_j
            if q < m[i,j]
                m[i,j] = q
    return m[i,j]                         //先计算后返回
```

# 自顶向下的备忘录版本 VS. 自底向上DP

- 自顶向下的备忘录版本
  - 只解决明确需要解决的子问题
  - 额外的递归调用开销
- 自底向上的动态规划
  - 所有的子问题必须被解决
  - 利用一些特定的表访问模式可以进一步降低算法的时间以及空间开销
- 两种方法都利用了重叠子问题的特性，时间复杂度都是 $O(n^3)$ ，初始化时有 $\Theta(n^2)$个子问题，每个子问题都只计算一次，但是每次计算都花费$O(n)$时间，故总共$O(n^3)$

# 小结

- 若所有子问题须至少解一次，自底向上的动态规划时间常数因子较优(不需要递归开销，维护表的开销较小)
- 若子问题空间有些不需要计算，则备忘型递归具有只需计算需要的子问题的优点。